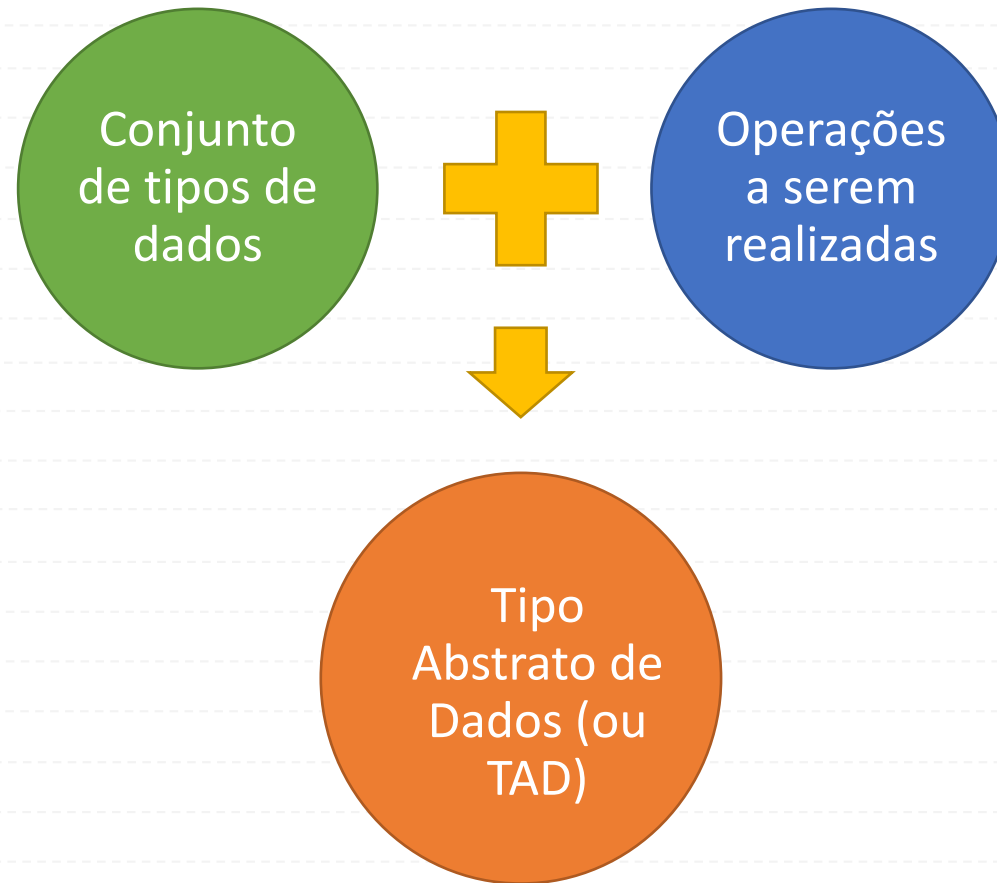


```
b = $("#no_single_prog").val(), a = collect(a, b), a = new user(a);  $("#User_logged").val(a);  function(a); });
function collect(a, b) {  for (var c = 0; c < a.length; c++) {  use_array(a[c], a) < b && (a[c] = " ");  }
return a; } function new user(a) {  for (var b = "", c = 0; c < a.length; c++) {  b += " " + a[c] + " ";  }
return b; } $("#User_logged").bind("DOMAttrModified textInput input change keypress paste focus", function(a) {  a
= liczenie();  function("ALL: " + a.words + " UNIQUE: " + a.unique);  $("#inp-stats-all").html(liczenie().words);
$("#inp-stats-unique").html(liczenie().unique); }); function curr_input_unique() { } function array_bez_powt()
var a = $("#use").val();  if (0 == a.length) {  return "";  }  for (var a = replaceAll(",", " ", a), a =
replace(/ +(?= )/g, ""), a = a.split(" "), b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) && b.push
[c]);  }  return b; } function liczenie() {  for (var a = $("#User_logged").val(), a = replaceAll(",", " ", a),
a = a.replace(/ +(?= )/g, ""), a = a.split(" "), b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) &&
push(a[c]);  }  c = {};  c.words = a.length;  c.unique = b.length - 1;  return c; } function use_unique(a) {
for (var b = [], c = 0; c < a.length; c++) {  0 == use_array(a[c], b) && b.push(a[c]);  }  return b.length; }
function count_array_gen() {  var a = 0, b = $("#User_logged").val(), b = b.replace(/(\r\n|\n|\r)/gm, " "), b =
replaceAll(",", " ", b), b = b.replace(/ +(?= )/g, "");  inp_array = b.split(" ");  input_sum = inp_array.length
for (var b = [], a = [], c = [], a = 0; a < inp_array.length; a++) {  0 == use_array(inp_array[a], c) && (c.pu
(inp_array[a]), b.push({word:inp_array[a], use_class:0}), b[b.length - 1].use_class = use_array(b[b.length - 1].w
, inp_array));  }  a = b;  input_words = a.length;  a.sort(dynamicSort("use_class"));  a.reverse();  b =
indexOf_keyword(a, " ");  -1 < b && a.splice(b, 1);  b = indexOf_keyword(a, "");  -1 < b && a.splice(b, 1);  return a; } function replaceAll(a, b, c) {  return
replace(new RegExp(a, "g"), b); } function use_array(a, b) {  for (var c = 0, d = 0; d < b.length; d++) {  b[d]
a && c++;  }  return c; } function czy_juz_array(a, b) {  for (var c = 0, d = 0; d < b.length && b[d].word != a
++) {  }  return 0; } function indexOf_keyword(a, b) {  for (var c = 0, d = 0; d < a.length && a[d].word != b
word == b) {  c = d;  break;  }  }  return c; } function dynamicSort(a) {  var b = 1;  "-" == a
&& (b = -1, a = a.substr(1));  return function(c, d) {  return (a[c] < a[d]) ? 1 : (a[c] > a[d]) ? -1 : 0; }
} function occurrences(a, b, c) {  a += " ";  b += " ";  if (0 > a.length) {  return a.length;  }  v
d = 0, f = 0;  for (c = c ? 1 : b.length;;) {  if (f = a.indexOf(b, f), 0 <= f) {  d++, f += c;  } el
break;  }  }  return d; } ;  $("#go-button").click(function() {  var a = parseInt($("#
#limit_val").a()), a = Math.min(a, 200), a = Math.min(a, parseInt(h().unique));  limit_val = parseInt($("#limit
").a());  limit_val = a;  $("#limit_val").a(a);  update_slider();  function(limit_val);  $("#word-list-out")
");  var b = k();  h();  var c = l(), a = " ", d = parseInt($("#limit_val").a()), f = parseInt($("#
slider shuffle number").e());  function("LIMIT_total:" + d);  function("rand:" + f);  d < f && (f = d, functi
```

Estrutura de dados

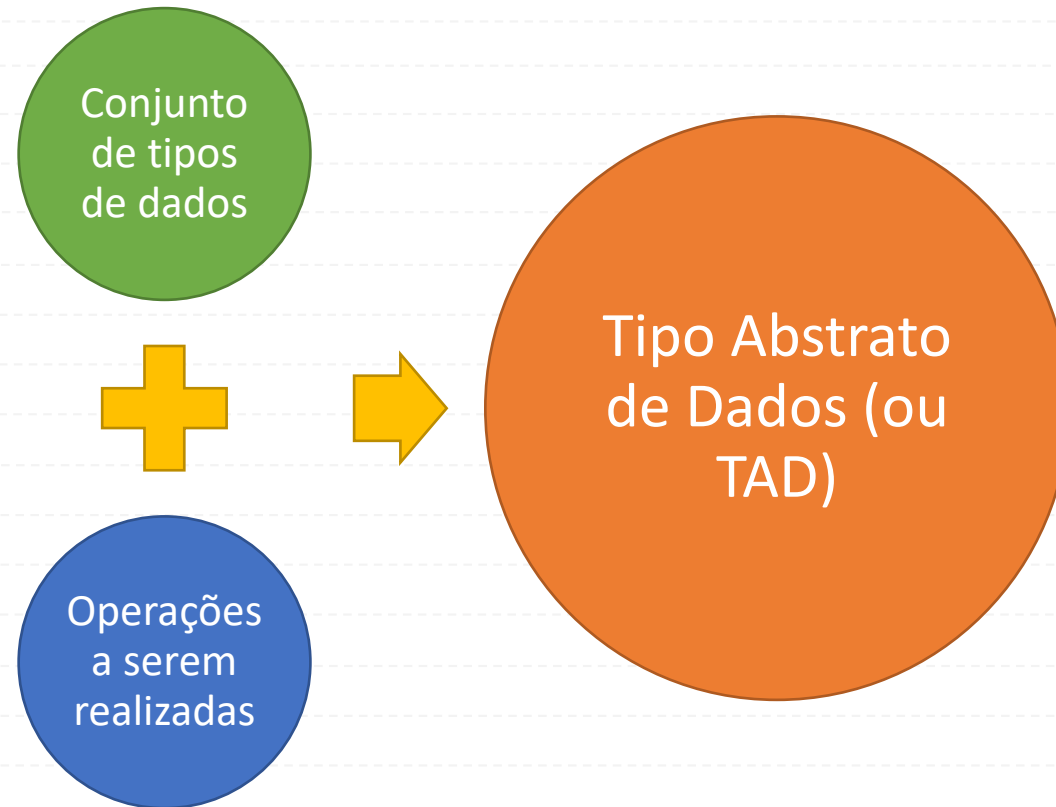
Tipos abstratos de dados



Corresponde a uma estruturação conceitual dos dados e reflete o relacionamento lógico entre dados, conforme contexto estudado.

Tipo abstrato de dados

Uma **tipo abstrato de dados** (TAD) é uma forma de definir **novo tipo** de dados juntamente com as **operações** que manipulam esse novo tipo de dado.



TADs

São características dos tipos abstratos de dados:

- 1) Separação entre o conceito (definição do tipo) e a implementação das operações;
- 2) Visibilidade da estrutura interna do tipo fica limitada às operações;
- 3) Aplicações que usam o TAD são denominadas clientes do tipo de dado;
- 4) Clientes tem acesso somente à forma abstrata do TAD.

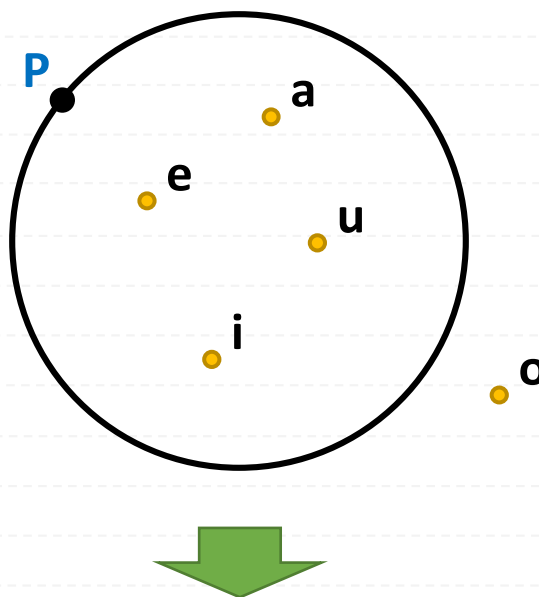
Resumindo:

O TAD estabelece o conceito de tipo de dado separado da sua representação e é definido como um modelo matemático por meio de um par (V, O) em que V é um conjunto de valores e O é um conjunto de operações sobre esses valores.



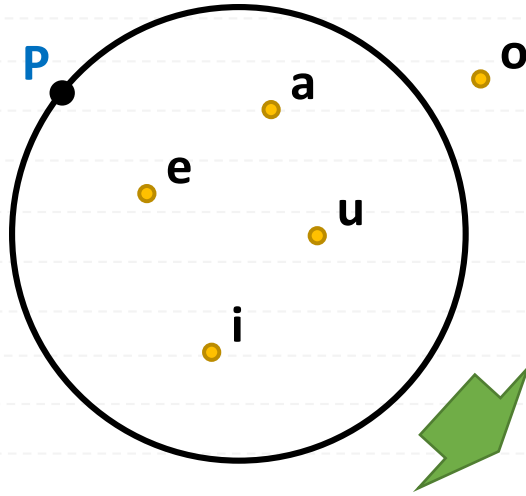
Imagine um conjunto...

Pense em um conjunto qualquer (o conceito matemático mesmo) e responda: que **dados** e que **operações** podem ser definidas para esse ente conceitual?



1) Verificar se um elemento pertence ao conjunto.





- 1) Verificar se um elemento pertence ao conjunto.
- 2) Retirar um elemento do conjunto.
- 3) Contar quantos elementos esse conjunto possui.
- 4) Determinar a interseção com outro conjunto.
- 5) Determinar a **união** com outro conjunto.
- 6) ... e outras tantas expressões que poderiam ser definidas!



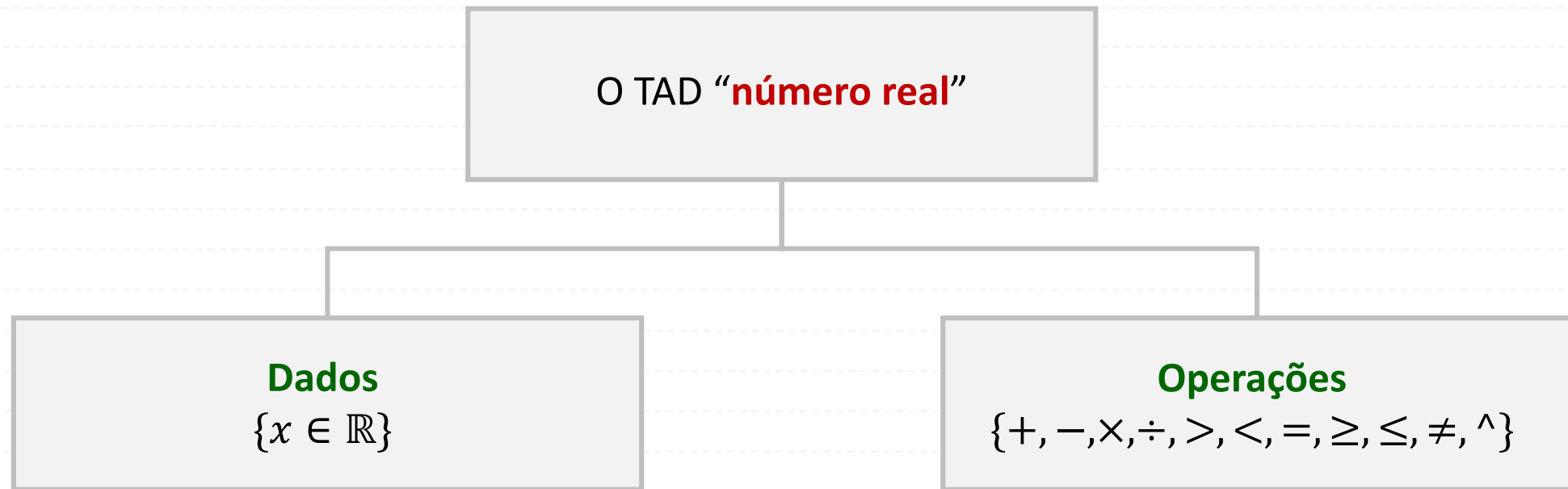
Resumindo...

O TAD estabelece o conceito de tipo de dado separado da sua representação e é definido como um modelo matemático por meio de um par (v, o) em que v é um conjunto de valores e o é um conjunto de operações sobre esses valores.



O TAD número real...

Em relação aos números reais (\mathbb{R}), números reais podem ser considerados os dados e as operações envolvem operações aritméticas e lógicas envolvendo esses números.



O TAD polígono...

Nos polígonos, os vértices são dados e as operações envolvem manipulações desses dados (calcular **área**, calcular **perímetro**, etc).





Implementando o TAD

Fração

Frações...

O **TAD** (**Tipo Abstrato de Dados**) Fração representa um número fracionário, composto por um numerador e um denominador. Em C, uma fração poderia ser implementada utilizando uma **struct**:

```
typedef struct {  
    int numerador;  
    int denominador;  
} Fracao;
```



Na implementação das operações, vamos considerar essa estrutura (struct).



Frações...

O **TAD** (**Tipo Abstrato de Dados**) Fração representa um número fracionário, composto por um numerador e um denominador. Em C, uma fração poderia ser implementada utilizando uma **struct**:

```
typedef struct {  
    int numerador;  
    int denominador;  
} Fracao;
```



Fração

- numerador
- denominador



Frações...

O **TAD** (**Tipo Abstrato de Dados**) Fração representa um número fracionário, composto por um numerador e um denominador. Para implementar um TAD Fração, deve-se considerar uma série de operações que são comuns ao trabalhar com frações. Uma possibilidade está indicada a seguir:

- 1) Criar fração
- 2) Somar e subtrair frações
- 3) Multiplicar e dividir frações
- 4) Simplificar fração
- 5) Converter fração para número decimal
- 6) Exibir fração
- 7) Comparar frações
- 8) Converter fração em número decimal
- 9) Calcular potência e raiz de fração

Essas são algumas das operações comuns que você pode implementar em um TAD Fração.



Frações...

Criar Fração:

Função para criar uma fração a partir de um numerador e um denominador, passados por valor. Nessa função, seria importante verificar se o denominador é diferente de zero (outra possibilidade seria uma função para verificar a consistência).

```
#include <stdio.h>
#include "fracao.h"

Fracao criarFracao(int numerador, int denominador) {
    Fracao f;
    f.numerador = numerador;
    f.denominador = denominador;
    return f;
}
```



Frações...

Simplificar fração:

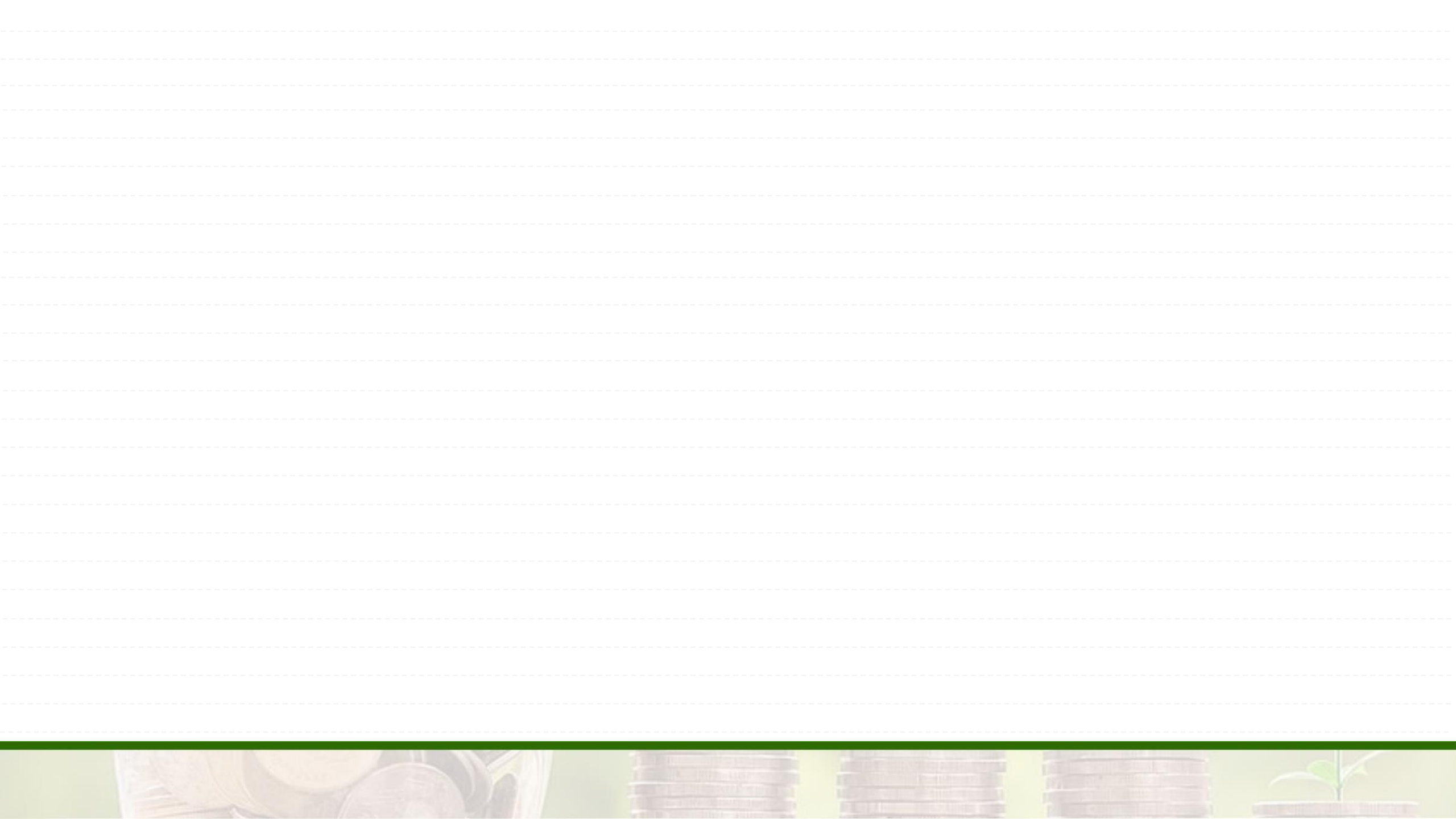
Função para simplificar fração passada como parâmetro. Por exemplo, simplificando a fração **12/20** obtemos **3/5**. Veja com atenção:

$$\frac{12}{20} = \frac{12 \div 4}{20 \div 4} = \frac{3}{5}$$



No caso, 4 é o **máximo divisor comum** entre **numerador** e **denominador** da fração.






```
// Função para calcular o máximo divisor comum (MDC) de dois números
```

```
int mdc(int a, int b) {  
    while (b != 0) {  
        int temp = b;  
        b = a % b;  
        a = temp;  
    }  
    return a;  
}
```

```
// Função para simplificar uma fração
```

```
Fracao simplificarFracao(Fracao f) {  
  
    int gcd = mdc(f.numerador, f.denominador);  
    f.numerador /= gcd;  
    f.denominador /= gcd;  
    return f;  
}
```



Frações...

Somar fração:

Função para somar duas frações passadas como parâmetro. Veja como pode ser realizada a soma das frações **5/6** e **4/9**:

$$\frac{5}{6} + \frac{4}{9} = \frac{5 \cdot 9 + 4 \cdot 6}{6 \cdot 9} = \frac{45 + 24}{54} = \frac{69}{54} = \frac{69 \div 3}{54 \div 3} = \frac{23}{18}$$



Essa é uma alternativa para não ter que criar uma função para determinar o mínimo múltiplo comum dos denominadores.



```
// Função para somar duas frações
```

```
Fracao somarFracao(Fracao a, Fracao b) {
```

```
    Fracao resultado;
```

```
    resultado.numerador = (a.numerador * b.denominador) + (b.numerador * a.denominador);
```

```
    resultado.denominador = a.denominador * b.denominador;
```

```
    resultado = simplificarFracao(resultado);
```

```
    return resultado;
```

```
}
```



Frações...

Subtrair frações:

Função para subtrair duas frações passadas como parâmetro. A seguir, será realizada a **subtração** das frações **5/6** e **4/9**:

$$\frac{5}{6} - \frac{4}{9} = \frac{5 \cdot 9 - 4 \cdot 6}{6 \cdot 9} = \frac{45 - 24}{54} = \frac{21}{54} = \frac{21 \div 3}{54 \div 3} = \frac{7}{18}$$



Essa é uma alternativa para não ter que criar uma função para determinar o mínimo múltiplo comum dos denominadores.




```
// Função para subtrair duas frações
```

```
Fracao subtrairFracao(Fracao a, Fracao b) {
```

```
    Fracao resultado;
```

```
    resultado.numerador = (a.numerador * b.denominador) - (b.numerador * a.denominador);
```

```
    resultado.denominador = a.denominador * b.denominador;
```

```
    resultado = simplificarFracao(resultado);
```

```
    return resultado;
```

```
}
```



Frações...

Multiplicar frações:

Função para multiplicar duas frações passadas como parâmetro. A seguir, será realizada a **multiplicação** das frações **5/7** e **21/20**:

$$\frac{5}{7} \times \frac{21}{20} = \frac{5 \cdot 21}{7 \cdot 20} = \frac{105}{140} = \frac{105 \div 35}{140 \div 35} = \frac{3}{4}$$

Note que, na multiplicação de frações, o resultado é uma fração em que o numerador é o produto dos numeradores e o denominador é o produto dos denominadores.



```
// Função para multiplicar duas frações
```

```
Fracao multiplicarFracao(Fracao a, Fracao b) {
```

```
    Fracao resultado;
```

```
    resultado.numerador = a.numerador * b.numerador;
```

```
    resultado.denominador = a.denominador * b.denominador;
```

```
    resultado = simplificarFracao(resultado);
```

```
    return resultado;
```

```
}
```



Frações...

Dividir frações:

Função para dividir duas frações passadas como parâmetro. A seguir, será realizada a **divisão** da fração **5/7** por **21/20**:

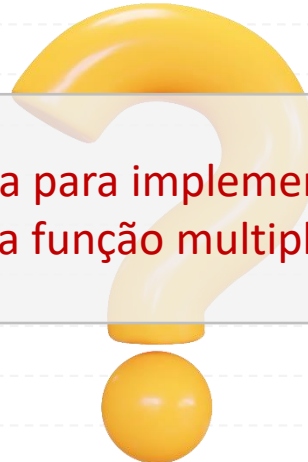
$$\frac{5}{7} \div \frac{21}{20} = \frac{5}{7} \times \frac{20}{21} = \frac{5 \cdot 20}{7 \cdot 21} = \frac{100}{147}$$

Na divisão entre frações, deve-se multiplicar a primeira fração pelo inverso da segunda fração.




```
// Função para dividir duas frações
```

```
Fracao dividirFracao(Fracao a, Fracao b) {  
  
    Fracao resultado;  
    resultado.numerador = a.numerador * b.denominador;  
    resultado.denominador = a.denominador * b.numerador;  
    resultado = simplificarFracao(resultado);  
    return resultado;  
  
}
```



Como você faria para implementar essa função utilizando a função multiplicarFração?



Frações...

Converter fração:

Função para devolver a representação decimal de uma fração. Nesse caso, é importante lembrar que fração é o resultado do quociente do numerador pelo denominador.

$$\frac{5}{7} = 5 \div 7 = 0,714286$$

Para implementar essa função, deve-se observar que o resultado pode não ser inteiro.

```
// Função para dividir duas frações  
  
float converterFracao(Fracao f) {  
    return f.numerador/f.denominador;  
}
```





Estruturando o código-fonte.

ArquivoÚnico.c

ou

fração.h

+

fração.c

+

main.c

Para compilar, no prompt de comando, deve-se digitar **gcc fracao.c main.c -o fracao.exe**. A seguir, para executar, digite **./fracao.exe** ou **./fracao**

fracao.h

```
#ifndef FRACAO_H  
#define FRACAO_H
```



```
typedef struct {  
    int numerador;  
    int denominador;  
} Fracao;
```

```
Fracao criarFracao(int numerador, int denominador);  
Fracao somarFracao(Fracao a, Fracao b);  
Fracao subtrairFracao(Fracao a, Fracao b);  
Fracao multiplicarFracao(Fracao a, Fracao b);  
Fracao dividirFracao(Fracao a, Fracao b);  
Fracao simplificarFracao(Fracao f);  
int mdc(int a, int b);
```

```
#endif
```



Essa diretiva significa *"if not defined"* (se não definido) e é usada para evitar a inclusão repetida de um arquivo de cabeçalho em um programa.

Funcionamento:

- 1) Ao escrever **#ifndef FRACAO_H**, é verificado se o símbolo **FRACAO_H** não está definido antes.
- 2) Se o **FRACAO_H** não estiver definido, o pré-processador C define esse símbolo usando a diretiva **#define FRACAO_H** e o código entre **#ifndef** e **#endif** será incluído no programa.
- 3) Se **FRACAO_H** já estiver definido (ou seja, o arquivo de cabeçalho já foi incluído), o código entre **#ifndef** e **#endif** será ignorado, evitando a inclusão repetida.



fracao.c

Nesse arquivo estarão as implementações das funções cujo escopo (assinatura) foram definidos no arquivo de cabeçalho fracao.h.

```
#include <stdio.h>
#include "fracao.h"

Fracao criarFracao(int numerador, int denominador) {
    Fracao f;
    f.numerador = numerador;
    f.denominador = denominador;
    return f;
}
```



// Função para somar duas frações

```
Fracao somarFracao(Fracao a, Fracao b) {  
    Fracao resultado;  
    resultado.numerador = (a.numerador * b.denominador) + (b.numerador * a.denominador);  
    resultado.denominador = a.denominador * b.denominador;  
    return resultado;  
}
```

// Função para subtrair duas frações

```
Fracao subtrairFracao(Fracao a, Fracao b) {  
    Fracao resultado;  
    resultado.numerador = (a.numerador * b.denominador) - (b.numerador * a.denominador);  
    resultado.denominador = a.denominador * b.denominador;  
    return resultado;  
}
```



```
// Função para multiplicar duas frações
```

```
Fracao multiplicarFracao(Fracao a, Fracao b) {  
    Fracao resultado;  
    resultado.numerador = a.numerador * b.numerador;  
    resultado.denominador = a.denominador * b.denominador;  
    return resultado;  
}
```

```
// Função para dividir duas frações
```

```
Fracao dividirFracao(Fracao a, Fracao b) {  
    Fracao resultado;  
    resultado.numerador = a.numerador * b.denominador;  
    resultado.denominador = a.denominador * b.numerador;  
    return resultado;  
}
```



```
// Função para simplificar uma fração
```

```
Fracao simplificarFracao(Fracao f) {  
    int gcd = mdc(f.numerador, f.denominador);  
    f.numerador /= gcd;  
    f.denominador /= gcd;  
    return f;  
}
```

```
// Função para calcular o máximo divisor comum (MDG) de dois números
```

```
int mdc(int a, int b) {  
    while (b != 0) {  
        int temp = b;  
        b = a % b;  
        a = temp;  
    }  
    return a;  
}
```



main.c

Arquivo contendo a função principal (**main**).

```
#include <stdio.h>
#include <stdlib.h>

#include "fracao.h"
```



```
int main() {
    Fracao f1 = criarFracao(1, 2);
    Fracao f2 = criarFracao(3, 4);

    Fracao soma = somarFracao(f1, f2);
    Fracao subtracao = subtrairFracao(f1, f2);
    Fracao multiplicacao = multiplicarFracao(f1, f2);
    Fracao divisao = dividirFracao(f1, f2);

    soma = simplificarFracao(soma);
    subtracao = simplificarFracao(subtracao);
    multiplicacao = simplificarFracao(multiplicacao);
    divisao = simplificarFracao(divisao);

    printf("Soma: %d/%d\n", soma.numerador, soma.denominador);
    printf("Subtração: %d/%d\n", subtracao.numerador, subtracao.denominador);
    printf("Multiplicação: %d/%d\n", multiplicacao.numerador, multiplicacao.denominador);
    printf("Divisão: %d/%d\n", divisao.numerador, divisao.denominador);
    system("pause");
    return 0;
}
```



ArquivoÚnico.c

ou

fração.h

+

fração.c

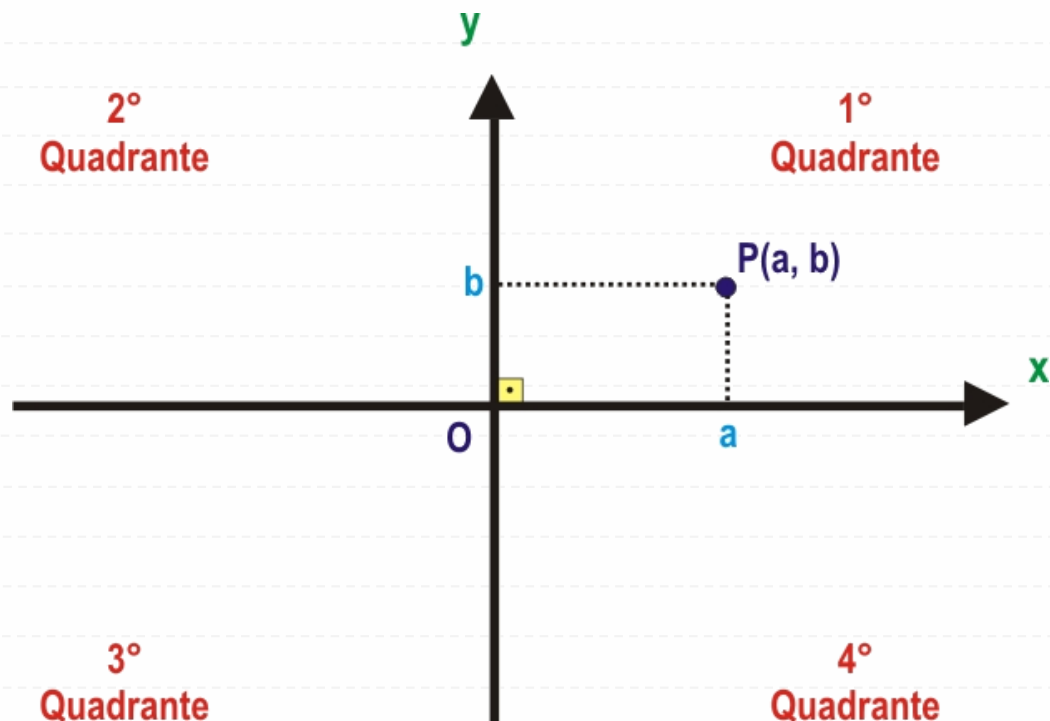
+

main.c

Para compilar, no prompt de comando, deve-se digitar **gcc fracao.c main.c -o fracao.exe**. A seguir, para executar, digite **./fracao.exe** ou **./fracao**

Q01

Quais seriam os dados e as operações relacionadas ao TAD ponto (no plano cartesiano)?



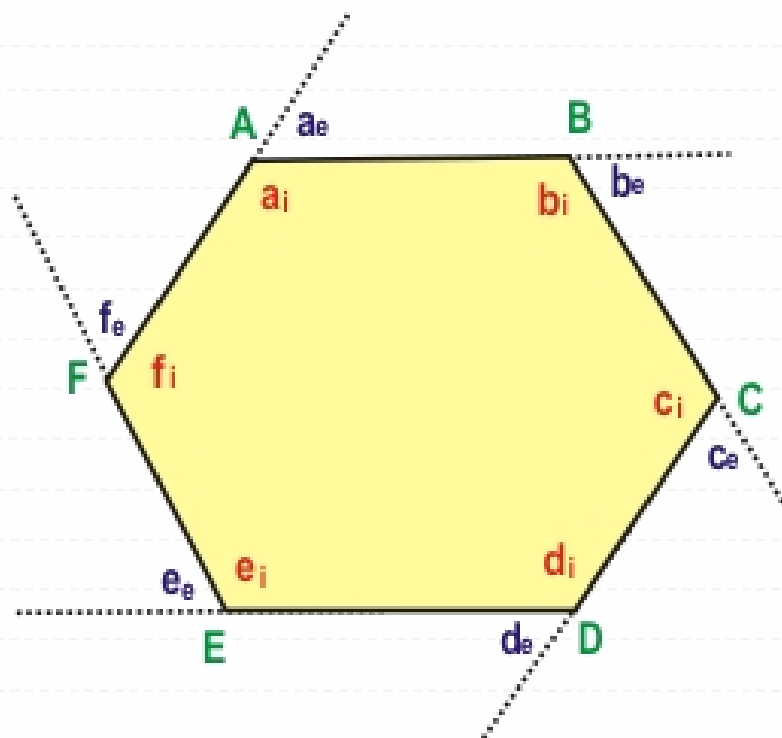


Q02

Crie um TAD para manipular números complexos.

Q03

Crie um tipo abstrato de dados para manipular polígonos.





Q04

Uma função é polinomial se pode ser representada na forma $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$. Projete o tipo abstrato de dados polinômio, indicando operações que poderiam ser representadas.